# CS422/522 Lab 1: Bootloader & Physical Memory Management due 2015-9-22

# Introduction

This lab is split into three parts. The first part concentrates on getting familiarized with x86 assembly language, the QEMU x86 emulator, and the PC's power-on bootstrap procedure. The second part examines the bootloader for our kernel, which resides in the `boot` directory of the `mcertikos` tree. The third part asks you to implement the physical memory management layers for the mCertiKOS kernel (we have provided an initial template in the `kernel` directory). Finally, there is an enrichment module which is totally optional (it will not be graded): it allows you to explore other files we provided in the initial mCertiKOS template.

Acknowledgments: this assignment is adapted from the CS422/522 labs developed by David Wolinsky in Fall 2014, which were in turn derived from the JOS labs in Frans Kaashoek's course 6.828 at MIT. The bootloader and the physical memory management interfaces used in this lab were developed by various members of the FLINT team at Yale.

## Software Setup

The files you will need for this and subsequent lab assignments in this course are distributed using the Git version control system. To learn more about Git, take a look at the Git user's manual, or, if you are already familiar with other version control systems, you may find this CS-oriented overview of Git useful.

The Git repository for the labs is in `/c/cs422/repo/mcertikos.git` on the zoo machines. However you need to set up your own repository to hand in each lab. To install the files, use the commands below with the desired location. You can log into Zoo remotely via ssh. To ssh to the zoo machines with window support, provide the -X option in the ssh command.

```
$ mkdir ~/cs422
$ cd ~/cs422
$ /c/cs422/apps/setrepo.sh mcertikos
Creating new repository /c/cs422/SUBMIT/lab/netid.git...
****
```

```
Now you can use 'git commit' and 'git push' to submit your code!
****
$ cd mcertikos
$
```

If you want to clone the repository remotely onto a non-Zoo machine after you have configured it as above, first get the path of your remote directory by running, from your local directory:

```
$ git remote show origin
* remote origin
  Fetch URL: /path/to/repo
  Push  URL: /path/to/repo
...
$
```

Suppose it is `/path/to/repo`. You can use `git clone` on your own computer:

```
$ git clone -b shell netid@node.zoo.cs.yale.edu:/path/to/repo lab
Cloning into lab...
$
```

Git allows you to keep track of the changes you make to the code. For example, if you are finished with one of the exercises, and want to checkpoint your progress, you can *commit* your changes by running:

```
$ git commit -am 'got test1 in lab1 working'
Created commit 60d2135: my solution for shell exercise
 1 files changed, 1 insertions(+), 0 deletions(-)
$
```

You can keep track of your changes by using the `git diff` command. Running `git diff` will display the changes to your code since your last commit, and `git diff origin/lab1` will display the changes relative to the initial code supplied for this lab. Here, `origin/lab1` is the name of the git branch with the initial code you downloaded from our server for this assignment.

## Getting Started

We have provided you with a starting point for your kernel source code. To fetch that source, fetch the latest version of the course repository, and then create a local branch called `lab1` based on our lab1 branch, `origin/lab1`:

```
$ cd ~/cpsc422/mcertikos
$ /c/cs422/apps/syncrepo.sh
****
Remote repository synchronized successfully.
****
$ git pull
Already up-to-date.
$ git checkout -b lab1 origin/lab1
Branch lab1 set up to track remote branch refs/remotes/origin/lab1.
Switched to a new branch "lab1"
$
```

The `/c/cs422/apps/syncrepo.sh` is a script which will synchronize your own remote repository with the master repository that we maintain for your assignments. Run the script everytime before you run `git pull`, in case there is any fixes we applied to existing code base. The `git checkout -b` command shown above actually does two things: it first creates a local branch `lab1` that is based on the `origin/lab1`, the course's branch, and second, it changes the contents of your `mcertikos` directory to reflect the files stored on the `lab1` branch. Git allows switching between existing branches using git checkout *branch-name*, though you should commit any outstanding changes on one branch before switching to a different one.

We have set up the appropriate compilers and simulators for you on the Zoo. To use them, run `source /c/cs422/env.sh`. You must run this command every time you log in (or add it to your `~/.environment` file). If you get obscure errors while compiling or running `qemu`, double check that you ran this command.

If you are working on a non-Zoo machine, you'll need to install `qemu` and possibly `gcc` following the directions on the tools page. If your machine uses a native ELF toolchain (such as Linux and most BSD's, but notably *not* OS X), you can simply install `gcc` from your package manager. Otherwise, follow the directions on the tools page.

* Simpler, you can always ssh to the one of the zoo machine and work remotely.

## Hand-In Procedure

Include the following information in the file called `README` in the `mcertikos` directory: who you have worked with, the answers to the questions in the **Question** section of this lab, brief description of what you have implemented. When you are ready to hand in your lab code, commit your changes, and then run `git push` in the `mcertikos`.

```
$ git commit -am "finished lab1"
[lab1 a823de9] finished lab1
 4 files changed, 87 insertions(+), 10 deletions(-)
$ git push
```

# Part 1: PC Bootstrap

The purpose of the first exercise is to introduce you to x86 assembly language and the PC bootstrap process, and to get you started with QEMU and QEMU/GDB debugging. You will not have to write any code for this part of the lab, but you should go through it anyway for your own understanding and be prepared to answer the questions posed below.

# Getting Started with x86 Assembly

If you are not already familiar with x86 assembly language, you will quickly become familiar with it during this course! The PC Assembly Language Book is an excellent place to start. Hopefully, the book contains mixture of new and old material for you.

*Warning:* Unfortunately the examples in the book are written for the NASM assembler, whereas we will be using the GNU assembler. NASM uses the so-called *Intel* syntax while GNU uses the *AT&T* syntax. While semantically equivalent, an assembly file will differ quite a lot, at least superficially, depending on which syntax is used. Luckily the conversion between the two is pretty simple, and is covered in Brennan's Guide to Inline Assembly.

> **Exercise 1**
>
> Familiarize yourself with the assembly language materials available on the reference page. You don't have to read them now, but you'll almost certainly want to refer to some of this material when reading and writing x86 assembly.

We do recommend reading the section "The Syntax" in Brennan's Guide to Inline Assembly. It gives a good (and quite brief) description of the AT&T assembly syntax, we'll be using with the GNU assembler in mCertiKOS.

Certainly the definitive reference for x86 assembly language programming is Intel's instruction set architecture reference, which you can find on the reference page in two flavors: an HTML edition of the old 80386 Programmer's Reference Manual, which is much shorter and easier to navigate than more recent manuals but describes all of the x86 processor features that we will make use of in CS422/522; and the full, latest and greatest IA-32 Intel Architecture Software Developer's Manuals from Intel, covering all the features of the most recent processors that we won't need in class but you may be interested in learning about. An equivalent (and often friendlier) set of manuals is available from AMD. Save the Intel/AMD architecture manuals for later or use them for reference when you want to look up the definitive explanation of a particular processor feature or instruction.

## Simulating the x86

Instead of developing the operating system on a real, physical personal computer (PC), we use a program that faithfully emulates a complete PC: the code you write for the emulator will boot on a real PC too. Using an emulator simplifies debugging; you can, for example, set break points inside of the emulated x86, which is difficult to do with the silicon version of an x86.

In CS422/522 we will use the QEMU Emulator, a modern and relatively fast emulator. While QEMU's built-in monitor provides only limited debugging support, QEMU can act as a remote debugging target for the GNU debugger (GDB), which we'll use in this lab to step through the early boot process.

To get started, extract the Lab 1 files into your own directory as described above in "Software Setup", then type `make` in the `mcertikos` directory to build the boot loader and kernel. (It's a little generous to call the code we're running here a "kernel," but we'll flesh it out throughout the semester.)

```
$ cd mcertikos
$ make
+ as[BOOT] boot/boot0/boot0.S
+ ld[BOOT] obj/boot/boot0
boot block is 131 bytes (max 446)
+ as[BOOT] boot/boot1/boot1.S
+ cc[BOOT] boot/boot1/boot1lib.c
+ cc[BOOT] boot/boot1/boot1main.c
+ as[BOOT] boot/boot1/exec_kernel.S
+ ld[BOOT] obj/boot/boot1
boot block is 5100 bytes (max 31744)
All targets of boot loader are done.
+ cc[KERN/dev] kern/dev/video.c
+ cc[KERN/dev] kern/dev/console.c
+ cc[KERN/dev] kern/dev/serial.c
+ cc[KERN/dev] kern/dev/devinit.c
+ cc[KERN/dev] kern/dev/mboot.c
+ cc[KERN/lib] kern/lib/string.c
+ cc[KERN/lib] kern/lib/debug.c
+ cc[KERN/lib] kern/lib/dprintf.c
+ cc[KERN/lib] kern/lib/printfmt.c
+ cc[KERN/lib] kern/lib/seg.c
+ cc[KERN/lib] kern/lib/types.c
+ cc[KERN/lib] kern/lib/x86.c
+ cc[KERN/init] kern/init/init.c
+ as[KERN/init] kern/init/entry.S
+ cc[KERN/pmm/MATIntro] kern/pmm/MATIntro/MATIntro.c
+ cc[KERN/pmm/MATInit] kern/pmm/MATInit/MATInit.c
+ cc[KERN/pmm/MATOp] kern/pmm/MATOp/MATOp.c
+ ld[KERN] obj/kern/kernel
All targets of kernel are done.
Building Certikos Image...

creating disk...
131040+0 records in
131040+0 records out
67092480 bytes (67 MB) copied, 0.182863 s, 367 MB/s
done.

writing mbr...
0+1 records in
0+1 records out
131 bytes (131 B) copied, 0.000322747 s, 406 kB/s
9+1 records in
9+1 records out
5100 bytes (5.1 kB) copied, 0.000342375 s, 14.9 MB/s
done.
```

```

copying kernel files...
kernel starts at sector 2048
44+1 records in
44+1 records out
22802 bytes (23 kB) copied, 0.00044239 s, 51.5 MB/s

All done.
All targets are done.
```

(If you get errors like "undefined reference to `__udivdi3'", you probably don't have the 32-bit gcc multilib. If you're running Debian or Ubuntu, try installing the gcc-multilib package.)

Now you're ready to run QEMU,supplying the file `certikos.img`, created above, as the contents of the emulated PC's "virtual hard disk." This hard disk image contains both our boot loader (`obj/boot`) and our kernel (`obj/kern`).

```
$ make qemu
```

* If you ssh'd to the zoo without the -X option, run `make qemu-nox` instead.

This executes QEMU with the options required to set the hard disk and direct serial port output to the terminal. Some text should appear in the QEMU window:

```
[D] kern/dev/devinit.c:14: cons initialized.
[D] kern/dev/devinit.c:15: devinit mbi_adr: 37260

BIOS-e820: 0x00000000 - 0x0009fbff (usable)
BIOS-e820: 0x0009fc00 - 0x0009ffff (reserved)
BIOS-e820: 0x000f0000 - 0x000fffff (reserved)
BIOS-e820: 0x00100000 - 0x7fffdfff (usable)
```

```
BIOS-e820: 0x7fffe000 - 0x7fffffff (reserved)
[D] kern/init/init.c:48: Kernel initialized.
[D] kern/init/init.c:18: In kernel main.



****************************************

Welcome to the mCertiKOS kernel monitor!

****************************************

Type 'help' for a list of commands.
$>
```

The `$>` is the prompt printed by the small *monitor*, or interactive control program, that we've included in the kernel. The lines printed by the kernel will also appear in the regular shell window from which you ran QEMU. This is because for testing and lab grading purposes we have set up the mCertiKOS kernel to write its console output not only to the virtual VGA display (as seen in the QEMU window), but also to the simulated PC's virtual serial port, which QEMU in turn outputs to its own standard output. Likewise, the mCertiKOS kernel will take input from both the keyboard and the serial port, so you can give it commands in either the VGA display window or the terminal running QEMU. Alternatively, you can use the serial console without the virtual VGA by running `make qemu-nox`. This may be convenient if you are SSH'd into the Zoo.

There are only two commands you can give to the kernel monitor, `help` and `kerninfo`.

```
$> help
help - display this list of commands
kerninfo - display information about the kernel
$> kerninfo
Special kernel symbols:
  start  001026e4
```

```
    etext   00102a47
    edata   00104528
    end     0014bb40
Kernel executable memory footprint: 294KB
$>
```

Since we have not set up the process management code, all the kernel does at the moment is to initialize all the devices and data structures, and jumps to the monitor. Although simple, it's important to note that this kernel is running "directly" on the "raw (virtual) hardware" of the simulated PC. This means that you should be able to copy the contents of `certikos.img` onto the first few sectors of a *real* hard disk, insert that hard disk into a real PC, turn it on, and see exactly the same thing on the PC's real screen as you did above in the QEMU window. (We don't recommend you do this on a real machine with useful information on its hard disk, though, because copying `certikos.img` onto the beginning of its hard disk will trash the master boot record and the beginning of the first partition, effectively causing everything previously on the hard disk to be lost!)

## The PC's Physical Address Space

We will now dive into a bit more detail about how a PC starts up. A PC's physical address space is hard-wired to have the following general layout:

```
+------------------+  <- 0xFFFFFFFF (4GB)
|      32-bit      |
|  memory mapped   |
|     devices      |
|                  |
/\/\/\/\/\/\/\/\/\/\

/\/\/\/\/\/\/\/\/\/\
|                  |
|      Unused      |
|                  |
+------------------+  <- depends on amount of RAM
|                  |
|                  |
| Extended Memory  |
|                  |
|                  |
+------------------+  <- 0x00100000 (1MB)
|     BIOS ROM     |
+------------------+  <- 0x000F0000 (960KB)
|  16-bit devices, |
|  expansion ROMs  |
+------------------+  <- 0x000C0000 (768KB)
|   VGA Display    |
+------------------+  <- 0x000A0000 (640KB)
|                  |
|    Low Memory    |
|                  |
+------------------+  <- 0x00000000
```

The first PCs, which were based on the 16-bit Intel 8088 processor, were only capable of addressing 1MB of physical memory. The physical address space of an early PC would therefore start at `0x00000000` but end at `0x000FFFFF` instead of `0xFFFFFFFF`. The 640KB area marked "Low Memory" was the *only* random-access memory (RAM) that an early PC could use; in fact the very earliest PCs could only be configured with 16KB, 32KB, or 64KB of RAM!

The 384KB area from `0x000A0000` through `0x000FFFFF` was reserved by the hardware for special uses such as video display buffers and firmware held in non-volatile memory. The most important part of this reserved area is the Basic Input/Output System (BIOS), which occupies the 64KB region from `0x000F0000` through `0x000FFFFF`. In early PCs the BIOS was held in true read-only memory (ROM), but current PCs store the BIOS in updateable flash memory. The BIOS is responsible for performing basic system initialization such as activating the video card and checking the amount of memory installed. After performing this initialization, the BIOS loads the operating system from some appropriate location such as floppy disk, hard disk, CD-ROM, or the network, and passes control of the machine to the operating system.

When Intel finally "broke the one megabyte barrier" with the 80286 and 80386 processors, which supported 16MB and 4GB physical address spaces respectively, the PC architects nevertheless preserved the original layout for the low 1MB of physical address space in order to ensure backward compatibility with existing software. Modern PCs therefore have a "hole" in physical memory from `0x000A0000` to `0x00100000`, dividing RAM into "low" or "conventional memory" (the first 640KB) and "extended memory" (everything else). In addition, some space at the very top of the PC's 32-bit physical address space, above all physical RAM, is now commonly reserved by the BIOS for use by 32-bit PCI devices.

Recent x86 processors can support *more* than 4GB of physical RAM, so RAM can extend further above `0xFFFFFFFF`. In this case the BIOS must arrange to leave a *second* hole in the system's RAM at the top of the 32-bit addressable region, to leave room for these 32-bit devices to be mapped. In mCertiKOS, we assume that all PCs have "only" a 32-bit physical address space (thus can support up to 4GB of memory). But dealing with complicated physical address spaces and other aspects of hardware organization that evolved over many years is one of the important practical challenges of OS development.

## The ROM BIOS

In this portion of the lab, you'll use QEMU's debugging facilities to investigate how an IA-32 compatible computer boots.

Open two terminal windows. If you ssh'd to the zoo, then make sure you ssh to the SAME zoo machine. In one, enter `make qemu-gdb` (or `make qemu-nox-gdb`). This starts up QEMU, but QEMU stops just before the processor executes the first instruction and waits for a debugging connection from GDB. In the second terminal, from the same directory you ran `make`, run `gdb`. You should see something like this,

```
$ gdb
GNU gdb (GDB) 6.8-debian
```

```
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
+ target remote localhost:1234
The target architecture is assumed to be i8086
[f000:fff0] 0xffff0:    ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
(gdb)
```

The following line:

```
[f000:fff0] 0xffff0:    ljmp   $0xf000,$0xe05b
```

is GDB's disassembly of the first instruction to be executed. From this output you can conclude a few things:

- The IBM PC starts executing at physical address `0x000ffff0`, which is at the very top of the 64KB area reserved for the ROM BIOS.
- The PC starts executing with `CS = 0xf000` and `IP = 0xfff0`.
- The first instruction to be executed is a `jmp` instruction, which jumps to the segmented address `CS = 0xf000` and `IP = 0xe05b`.

Why does QEMU start like this? This is how Intel designed the 8088 processor, which IBM used in their original PC. Because the BIOS in a PC is "hard-wired" to the physical address range `0x000f0000-0x000fffff`, this design ensures that the BIOS always gets control of the machine first after power-up or any system restart - which is crucial because on power-up there *is* no other software anywhere in the machine's RAM that the processor could execute. The QEMU emulator comes with its own BIOS, which it places at this location in the processor's simulated physical address space. On processor reset, the (simulated) processor enters real mode and sets CS to `0xf000` and the IP to `0xfff0`, so that execution begins at that (CS:IP) segment address. How does the segmented address 0xf000:fff0 turn into a physical address?

To answer that we need to know a bit about real mode addressing. In real mode (the mode that PC starts off in), address translation works according to the formula: *physical address* = 16 * *segment* + *offset*. So, when the PC sets CS to `0xf000` and IP to `0xfff0`, the physical address referenced is:

```
16 * 0xf000 + 0xfff0      # in hex multiplication by 16 is
   = 0xf0000 + 0xfff0     # easy--just append a 0.
   = 0xffff0
```

`0xffff0` is 16 bytes before the end of the BIOS (`0x100000`). Therefore we shouldn't be surprised that the first thing that the BIOS does is `jmp` backwards to an earlier location in the BIOS; after all how much could it accomplish in just 16 bytes?

When the BIOS runs, it sets up an interrupt descriptor table and initializes various devices such as the VGA display. After initializing the PCI bus and all the important devices the BIOS knows about, it searches for a bootable device such as a floppy, hard drive, or CD-ROM. Eventually, when it finds a bootable disk, the BIOS reads the *boot loader* from the disk and transfers control to it.

---

## Part 2: Bootloader

Floppy and hard disks for PCs are divided into 512 byte regions called *sectors*. A sector is the disk's minimum transfer granularity: each read or write operation must be one or more sectors in size and aligned on a sector boundary. If the disk is bootable, the first sector is called the *boot sector*, since this is where the boot loader code resides. When the BIOS finds a bootable floppy or hard disk, it loads the 512-byte boot sector into memory at physical addresses `0x7c00` through `0x7dff`, and then uses a `jmp` instruction to set the CS:IP to `0000:7c00`, passing control to the boot loader. Like the BIOS load address, these addresses are fairly arbitrary - but they are fixed and standardized for PCs.

The ability to boot from a CD-ROM came much later during the evolution of the PC, and as a result the PC architects took the opportunity to rethink the boot process slightly. As a result, the way a modern BIOS boots from a CD-ROM is a bit more complicated (and more powerful). CD-ROMs use a sector size of 2048 bytes instead of 512, and the BIOS can load a much larger boot image from the disk into memory (not just one sector) before transferring control to it. For more information, see the "El Torito" Bootable CD-ROM Format Specification.

However, we will use the conventional hard drive boot mechanism, which means that our boot loader must either fit into a measly 512 bytes, or split into two parts, and let the first part load the second one. We will choose the second method, because it enables us to design a fancier boot loader.

The first part is called `BOOT0`, which is in the assembly source file `boot/boot0/boot0.S`. It does one job: to load the `BOOT1` from sector 2 ~ 63 on the disk into physical memory `0000:7e00` through `0000:f9ff`. `BOOT1` is the *"real"* boot loader. It consists of 4 files:

- `boot/boot1/boot1.S` assembly part of `BOOT1`
- `boot/boot1/boot1main.c` C part of `BOOT1`
- `boot/boot1/boot1lib.c` auxiliary functions
- `boot/boot1/exec_kernel.S` setting the arguments, and jumping to kernel

Look through these source files carefully and make sure you understand what's going on. The boot loader must perform three main functions:

1. First, the boot loader switches the processor from real mode to *32-bit protected mode*, because it is only in this mode that software can access all the memory above 1MB in the processor's physical address space. Protected mode is described briefly in sections 1.2.7 and 1.2.8 of PC Assembly Language, and in great detail in the Intel architecture manuals. At this point, you only have to understand that translation of segmented addresses (segment:offset pairs) into physical addresses happens differently in protected mode, and that after the transition offsets are 32 bits instead of 16.
2. Second, the boot loader finds and reads the kernel from the hard disk by directly accessing the IDE disk device registers via the x86's special I/O instructions. If you would like to understand better what the particular I/O instructions here mean, check out the "IDE hard drive controller" section on the reference page. You will not need to learn much about programming specific devices in this class: writing device

drivers is in practice a very important part of OS development, but from a conceptual or architectural viewpoint it is also one of the least interesting.
Our kernel starts at the first bootable partition. To understand the concept of `bootable` and `partition` you can read the reference page.

3. Third, the boot loader detects the physical memory mapping information and passes it to kernel. To know which address of RAM is available and which is reserved by BIOS or other devices is vital for a kernel, because writing to a reserved memory location may cause system fault or even hardware damage. To detect physical memory mapping, you have to use `BIOS functions`. These functions can only be accessed in the real mode, but kernel usually runs in 32-bit protected mode. As a result, it becomes boot loader's job to detect physical memory mapping.
   `BIOS functions` are organized by software interrupts (`INT` instruction), with a command number (stored in `%eax`):

   - INT 0x10 = Video display functions (including VESA/VBE)
   - INT 0x13 = mass storage (disk, floppy) access
   - INT 0x15 = memory size functions
   - INT 0x16 = keyboard functions

   Invoking `INT 0x15` with `%eax` being set to `0x0e820` means to get complete memory map. Thus, "e820" becomes the name of physical memory map table. The details of detecting e820 table can be find in the osdev page.

```

                                           Memory
                                     +-------------------+   <- 0x0000fa00
                +------------------  | [     BOOT1     ] |   <- 0x00007e00
                |                    +-------------------+
           +----(--(BIOS)----------  | [     BOOT0     ] |   <- 0x00007c00
           |    |                    +-------------------+
           |    |                    | [     stack     ] |
           |    |                    |                   |
           |    |                    +------------------+  <- 0x00000500
           |  /------------\
         +---+---+---\  /---+---\   /---+
   Disk  |MBR|   |   /  \   |   /   \   |
         |   |   |   \  /   |   \   /   |
         +---+---+---/  \---+---/   \---+
 Sector    1   2   3      63  64 ...
```

After you understand the boot loader source code, look at the file `obj/boot/boot0.asm` and `obj/boot/boot1.asm`. These files are disassemblies of the boot loader that our Makefile creates *after* compiling the boot loader. These disassembly files make it easy to see exactly where in physical memory all of the boot loader's code resides, and makes it easier to track what's happening while stepping through the boot loader in GDB. Likewise, `obj/kern/kernel.asm` contains a disassembly of the mCertiKOS kernel, which can often be useful for debugging.

You can set address breakpoints in GDB with the `b` command. For example, `b *0x7c00` sets a breakpoint at address `0x7C00`. Once at a breakpoint, you can continue execution using the `c` and `si` commands: `c` causes QEMU to continue execution until the next breakpoint (or until you press Ctrl-C in GDB), and `si N` steps through the instructions `N` at a time.

To examine instructions in memory (besides the immediate next one to be executed, which GDB prints automatically), you use the `x/i` command. This command has the syntax `x/Ni ADDR`, where `N` is the number of consecutive instructions to disassemble and `ADDR` is the memory address at which to start disassembling.

## Exercise 2

Take a look at the lab tools guide, especially the section on GDB commands. Even if you're familiar with GDB, this includes some esoteric GDB commands that are useful for OS work.

Set a breakpoint at address 0x7e00, which is where the boot sector will be loaded. Continue execution until that breakpoint. Trace through the code in `boot/boot1/boot1.S`, using the source code and the disassembly file `obj/boot/boot1.asm` to keep track of where you are. Also use the `x/i` command in GDB to disassemble sequences of instructions in the boot loader, and compare the original boot loader source code with both the disassembly in `obj/boot/boot1.asm` and GDB. You can also you the command `layout asm` to show the disassembled instruction list; and use the command `layout regs` to show the values of the current registers.

Set a break point to `boot1main` and trace into `load_kernel()` in `boot/boot1/boot1main.c`, and then into `readsection()`. Once you are in a code section corresponding to a compiled C code, you can run `layout asm`, then `list` to load the original C source code. Then you can run `next` to execute the code line by line at the C level.

> **Question A**
>
> 1. At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?
> 2. What is the *last* instruction of the boot loader executed, and what is the *first* instruction of the kernel it just loaded?
> 3. *Where* is the first instruction of the kernel?
> 4. How does the boot loader decide how many sectors it must read in order to fetch the entire kernel from disk? Where does it find this information?

## Loading the Kernel

We will now look in further detail at the C language portion of the boot loader, in `boot/boot1/boot1main.c`.

To make sense out of `boot/boot1/boot1main.c` you'll need to know what an ELF binary is. When you compile and link a C program such as the mCertiKOS kernel, the compiler transforms each C source ('`.c`') file into an *object* ('`.o`') file containing assembly language instructions encoded in the binary format expected by the hardware. The linker then combines all of the compiled object files into a single *binary image* such as `obj/kern/kernel`, which in this case is a binary in the ELF (Executable and Linkable Format) format.

Full information about this format is available in the ELF specification, but you will not need to delve very deeply into the details of this format in this class. Although as a whole the format is quite powerful and complex, most of the complex parts are for supporting dynamic loading of shared libraries, which we will not do in this class.

For purposes of this class, you can consider an ELF executable to be a header with loading information, followed by several *program sections*, each of which is a contiguous chunk of code or data intended to be loaded into memory at a specified address. The boot loader does not modify the code or data; it loads it into memory and starts executing it.

An ELF binary starts with a fixed-length *ELF header*, followed by a variable-length *program header* listing each of the program sections to be loaded. The C definitions for these ELF headers are in `boot/boot1/boot1lib.h (line 132)`. The program sections we're interested in are:

- `.text`: The program's executable instructions.
- `.rodata`: Read-only data, such as ASCII string constants produced by the C compiler. (We will not bother setting up the hardware to prohibit writing, however.)
- `.data`: The data section holds the program's initialized data, such as global variables declared with initializers like `int x = 5;`.

When the linker computes the memory layout of a program, it reserves space for *uninitialized* global variables, such as `int x;`, in a section called `.bss` that immediately follows `.data` in memory. C requires that "uninitialized" global variables start with a value of zero. Thus there is no need to store contents for `.bss` in the ELF binary; instead, the linker records just the address and size of the `.bss` section. The loader or the program itself must arrange to zero the `.bss` section.

Examine the full list of the names, sizes, and link addresses of all the sections in the kernel executable by typing:

```
$ objdump -h obj/kern/kernel
```

You will see many more sections than the ones we listed above, but the others are not important for our purposes. Most of the others are to hold debugging information, which is typically included in the program's executable file but not loaded into memory by the program loader.

Take particular note of the "VMA" (or *link address*) and the "LMA" (or *load address*) of the `.text` section. The load address of a section is the memory address at which that section should be loaded into memory. In the ELF object, this is stored in the `ph->p_pa` field (in this case, it really is a physical address, though the ELF specification is vague on the actual meaning of this field).

The link address of a section is the memory address from which the section expects to execute. The linker encodes the link address in the binary in various ways, such as when the code needs the address of a global variable, with the result that a binary usually won't work if it is executing from an address that it is not linked for. (It is possible to generate *position-independent* code that does not contain any such absolute addresses. This is used extensively by modern shared libraries, but it has performance and complexity costs, so we won't be using it in this class.)

Typically, the link and load addresses are the same. For example, look at the `.text` section of the boot loader:

```
$ objdump -h obj/boot/boot0.elf
```

The BIOS loads the boot sector into memory starting at address 0x7c00, so this is the boot sector's load address. This is also where the boot sector executes from, so this is also its link address. We set the link address by passing `-Ttext 0x7C00` to the linker in `boot/boot0/Makefile.inc`, so the linker will produce the correct memory addresses in the generated code.

> ### Exercise 3
>
> Trace through the first few instructions of the boot loader again and identify the first instruction that would "break" or otherwise do the wrong thing if you were to get the boot loader's link address wrong. Then change the link address in

> `boot/boot0/Makefile.inc` to something wrong, run make clean, recompile the lab with make, and trace into the boot loader again to see what happens. Don't forget to change the link address back and make clean again afterward!

Besides the section information, there is one more field in the ELF header that is important to us, named `e_entry`. This field holds the link address of the *entry point* in the program: the memory address in the program's text section at which the program should begin executing. You can see the entry point:

```
$ objdump -f obj/kern/kernel
```

You should now be able to understand the minimal ELF loader in `boot/boot1/boot1main.c`. It reads each section of the kernel from disk into memory at the section's load address and then jumps to the kernel's entry point.

> ### Exercise 4
>
> We can examine memory using GDB's x command. The GDB manual has full details, but for now, it is enough to know that the command `x/Nx ADDR` prints `N` words of memory at `ADDR`. (Note that both `x`s in the command are lowercase.) *Warning*: The size of a word is not a universal standard. In GNU assembly, a word is two bytes (the 'w' in xorw, which stands for word, means 2 bytes).

---

# Part 3: Physical Memory Management

The operating system must keep track of which parts of physical RAM are free and which are currently in use. In this part, you will implement the physical memory management code for mCertiKOS, so that the kernel can allocate memory and later free it. Your allocator will operate in units of 4096 bytes, called *pages*. mCertiKOS manages the PC's physical memory with *page granularity* so that it can use the MMU to map and protect each piece of allocated memory. Your task will be to maintain data structures that record which physical pages are reserved by BIOS, which are free, and which are already allocated by the kernel. You will also write the routines to allocate and free pages of memory.

You will implement various parts of the physical memory management module strictly following the abstraction layers that we have built for you. Inside the `kern` directory, you will see a sub directory called `pmm`. This is where all the code related to physical memory management reside. The physical memory management is devided into three abstraction layers, which corresponds the three further sub directories you see under `pmm`. You will need to implement layer by layer, from bottom up, following the descriptions below. Each layer directory contains the following three fires:

- `import.h:` The list of functions that are exposed to the current layer are declared and documented here. You are supposed to implement the layer functions using only the functions declared in import.h. This way, you do not have to look at the lower layers to figure out all the details.
- `LayerName.c` The list of functions in the current layer are implemented here. You are supposed to fill in the part marked as `TODO`.
- `export.h` The declarations of the functions of the current layer that are exposed to the upper layers.

## The MATIntro Layer

In this layer, you are going to implement getter and setter functions for a data structure used to store the physical memory information. In `MATIntro.c`, the data structure `AT` is a simple array of a physical allocation table entry structure that stores the permission and allocation status of each physical page. Please make sure you read all the comments carefully.

> ### Exercise 5
>
> In the file `kern/pmm/MATIntro/MATIntro.c`, you must implement all the functions listed below:

> - `at_is_norm`
> - `at_set_perm`
> - `at_is_allocated`
> - `at_set_allocated`

## Testing The Kernel

We will be grading your code with a bunch of test cases, part of which are given in `test.c` in each layer sub directory. You can run `make TEST=1` to test your solutions. You can use `Ctrl-a x` to exit from the qemu.

* If you have already run `make` before, you have to first run `make clean` before you run `make TEST=1`.

```
Welcome to mCertiKOS!

Testing the MATIntro layer...
test 1 passed.
test 2 failed.
test 3 failed.
Test failed.

Testing the MATInit layer...
test 1 failed.
Test failed.

Testing the MATOp layer...
test 1 failed.
Test failed.
```

Make sure your code passes all the tests for the `MATIntro` layer.

### Write Your Own Test Cases! (optional)

Come up with your own interesting test cases to seriously challenge your classmates! In addition to the provided simple tests, selected (correct, fully documented, and interesting) test cases will be used in the actual grading of the lab assignment!

In `test.c` in each layer directory, you will find a function defined with the name `LayerName_test_own`. Fill the function body with all of your nice test cases combined. The test function should return 0 for passing the test and a non-zero code for failing the test. Be extra careful to make sure that if you overwrite some of the kernel data, they are set back to the original value. O.w., it may make the future test scripts to fail even if you implement all the functions correctly.

* Your test function itself will not be graded. So don't be afraid of submitting a wrong script.

## The MATInit Layer

In this layer, you will implement an initilization function that detects and sets the maximum number of pages available in the machine, and initilizes the allocation table `AT` you've just implemented in the lower layer (`MATIntro`).

During the start up, the boot loader will load from BIOS the information on which memory ranges are reserved by BIOS, which are available. Please read the comments in `import.h` very carefully on the set of functions that can be used to retrieve this part of information. In addition, our mCertiKOS kernel reserves the physical address `0` to `1GB` for the kernel memory. More detailed documentations and hints can be found in the comments in `MATInit.c`.

> **Exercise 6**
>
> In the file `kern/pmm/MATInit/MATInit.c`, you must correctly implement the function:
>
> - `pmem_init`

Make sure your code passes all the tests for the `MATInit` layer. And write your own test cases to challenge other students' implementations.

## The MATOp Layer

In this layer, you will implement the functions to allocate and free pages of memory. Please review the list of functions you may need in `import.h`. In `MATOp.c`, you are asked to implement a fairly naive version of a physical page allocator and deallocator, and then come up with a slightly optimized version using the memorization. Please review the comments carefully for more details.

> ### Exercise 7
>
> In the file `kern/pmm/MATOp/MATOp.c`, you must correctly implement all the functions listed below:
>
> - `palloc`
> - `pfree`

Make sure your code passes all the tests for the `MATOp` layer. And write your own test cases to challenge other students' implementations.

**This completes the lab.** Make sure you pass all of the `make TEST=1` tests and don't forget editing your `README` file. Commit your changes and type `git push` in the `mcertikos` directory to submit your code.

---

# Enrichment (optional)

(This part will not be graded)

## Formatted Printing to the Console

Most people take functions like `printf()` for granted, sometimes even thinking of them as "primitives" of the C language. But in an OS kernel, we have to implement all I/O ourselves.

Read through `kern/lib/dprintf.c`, `kern/lib/printfmt.c`, `kern/dev/console.c`, `kern/dev/serial.c`, `kern/dev/video.c`, and `kern/dev/keyboard.c`, and make sure you understand their relationship.

### Exercise 8

We have omitted a small fragment of code - the code necessary to print octal numbers using patterns of the form "%o". Find and fill in this code fragment.

To test code, per questions 3, 4, and 5, one option is to modify monitor.c and insert one or more additional commands.

### Question B

1. Explain the interface between `dprintf.c` and `console.c`. Specifically, what function does `console.c` export? How is this function used by `dprintf.c`?
2. Explain the following from `video.c`:

```
if (terminal.crt_pos >= CRT_SIZE) {
    int i;
    memmove(terminal.crt_buf, terminal.crt_buf + CRT_COLS,
        (CRT_SIZE - CRT_COLS) * sizeof(uint16_t));
    for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
        terminal.crt_buf[i] = 0x0700 | ' ';
        terminal.crt_pos -= CRT_COLS;
}
```

3. Explore GCC's calling convention on the x86, and trace the execution of the following code step-by-step:

```
int x = 1, y = 3, z = 4;
dprintf("x %d, y %x, z %d\n", x, y, z);
```

   1. In the call to `dprintf()`, to what does `fmt` point? To what does `ap` point?
   2. List (in order of execution) each call to `cons_putc`, `va_arg`, and `vdprintf`. For `cons_putc`, list its argument as well. For `va_arg`, list what `ap` points to before and after the call. For `vdprintf` list the values of its two arguments.

4. Run the following code.

```
unsigned int i = 0x00646c72;
dprintf("H%x Wo%s", 57616, &i);
```

   What is the output? Explain how this output is arrived at in the step-by-step manner of the previous exercise. Here's an ASCII table that maps bytes to characters.

   The output depends on that fact that the x86 is little-endian. If the x86 were instead big-endian what would you set `i` to in order to yield the same output? Would you need to change `57616` to a different value?

   Here's a description of little- and big-endian and a more whimsical description.

5. In the following code, what is going to be printed after `y=`? (note: the answer is not a specific value.) Why does this happen?

```
dprintf("x=%d y=%d", 3);
```

6. Let's say that GCC changed its calling convention so that it pushed arguments on the stack in declaration order, so that the last argument is pushed last. How would you have to change `dprintf` or its interface so that it would still be possible to pass it a variable number of arguments?

## The Stack

In the final exercise of this lab, we will explore in more detail the way the C language uses the stack on the x86, and in the process write a useful new kernel monitor function that prints a *backtrace* of the stack: a list of the saved Instruction Pointer (IP) values from the nested `call` instructions that led to the current point of execution.

The x86 stack pointer (`esp` register) points to the lowest location on the stack that is currently in use. Everything *below* that location in the region reserved for the stack is free. Pushing a value onto the stack involves decreasing the stack pointer and then writing the value to the place the stack pointer points to. Popping a value from the stack involves reading the value the stack pointer points to and then increasing the stack pointer. In 32-bit mode, the stack can only hold 32-bit values, and esp is always divisible by four. Various x86 instructions, such as `call`, are "hard-wired" to use the stack pointer register.

The `ebp` (base pointer) register, in contrast, is associated with the stack primarily by software convention. On entry to a C function, the function's *prologue* code normally saves the previous function's base pointer by pushing it onto the stack, and then copies the current `esp` value into `ebp` for the duration of the function. If all the functions in a program obey this convention, then at any given point during the program's execution, it is possible to trace back through the stack by following the chain of saved `ebp` pointers and determining exactly what nested sequence of function calls caused this particular point in the program to be reached. This capability can be particularly useful, for example, when a particular function causes an `assert` failure or `panic` because bad arguments were passed to it, but you aren't sure *who* passed the bad arguments. A stack backtrace lets you find the offending function.

The above information should give you some idea on how to implement a stack backtrace function, which you should call `mon_backtrace()`. A prototype for this function is already waiting for you in `kern/lib/monitor.c`. You can do it entirely in C, but you may find the `read_ebp()` function in `kern/lib/x86.h` useful. You'll also have to hook this new function into the kernel monitor's command list so that it can be invoked interactively by the user.

The backtrace function should display a listing of function call frames in the following format:

```
Stack backtrace:
  ebp f0109e58  eip f0100a62  args 00000001 f0109e80 f0109e98 f0100ed2 00000031
  ebp f0109ed8  eip f01000d6  args 00000000 00000000 f0100058 f0109f28 00000061
  ...
```

The first line printed reflects the *currently executing* function, namely `mon_backtrace` itself, the second line reflects the function that called `mon_backtrace`, the third line reflects the function that called that one, and so on. You should print *all* the outstanding stack frames. By studying `kern/init/entry.S` you'll find that there is an easy way to tell when to stop.

Within each line, the `ebp` value indicates the base pointer into the stack used by that function: i.e., the position of the stack pointer just after the function was entered and the function prologue code set up the base pointer. The listed `eip` value is the function's *return instruction pointer*: the instruction address to which control will return when the function returns. The return instruction pointer typically points to the instruction after the `call` instruction (why?). Finally, the five hex values listed after `args` are the first five arguments to the function in question, which would have been pushed on the stack just before the function was called. If the function was called with fewer than five arguments, of course, then not all five of these values will be useful. (Why can't the backtrace code detect how many arguments there actually are? How could this limitation be fixed?)

Here are a few specific points that are worth remembering for the following exercise and for future labs.

- If `int *p = (int*)100`, then `(int)p + 1` and `(int)(p + 1)` are different numbers: the first is `101` but the second is `104`. When adding an integer to a pointer, as in the second case, the integer is implicitly multiplied by the size of the object the pointer points to.
- `p[i]` is defined to be the same as `*(p+i)`, referring to the i'th object in the memory pointed to by p. The above rule for addition helps this definition work when the objects are larger than one byte.
- `&p[i]` is the same as `(p+i)`, yielding the address of the i'th object in the memory pointed to by p.

Although most C programs never need to cast between pointers and integers, operating systems frequently do. Whenever you see an addition involving a memory address, ask yourself whether it is an integer addition or pointer addition and make sure the value being added is appropriately multiplied or not.

## Exercise 9

Implement the backtrace function as specified above, using the same format as in the example. Then add a `backtrace` command to the kernel monitor:

```
$> backtrace
Stack backtrace:
  ebp f010ff78  eip f01008ae  args 00000001 f010ff8c 00000000 f0110580 00000000
  ebp f010ffd8  eip f0100193  args 00000000 00001aac 00000660 00000000 00000000
  ebp f010fff8  eip f010003d  args 00000000 00000000 0000ffff 10cf9a00 0000ffff
$>
```